Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5707--02-8604

RASE — Run-time Automated Schema Evolution

An Implementation of Schema Evolution and Mapping

Brian Solan
Gregory Stern
Michael Pilone

Effectiveness of Naval Electronic Warfare Systems
Tactical Electronic Warfare Division

April 15, 2002

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
April 15, 2002	Memorandum Report	March 2000-March 2001

4. TITLE AND SUBTITLE

RASE—Run-time Automated Schema Evolution
An Implementation of Schema Evolution and Mapping

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Brian Solan, Gregory Stern, and Michael Pilone

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Tactical Electronic Warfare Division
4555 Overlook Ave., SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5707--02-8604

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR / MONITOR'S ACRONYM(S)

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software needs and requirements change over time as application developers fine-tune and expand the features of a software product. When the software is changed to meet these new requirements, some data structures need to be added and existing ones need to be altered or evolved. This paper describes an implementation of a sophisticated, dynamically evolving schema. The schema definition supports single and multiple inheritance, fixed length arrays, built-in strings, instances of objects and pointers to other objects. The sample implementation in this report is geared towards an object database.

15. SUBJECT TERMS

Schema, schema evolution, schema mapping, schema registering, run-time automated schema evolution, RASE, database, object database

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James G. Durbin
Unclassified	Unclassified	Unclassified	UL	32	19b. TELEPHONE NUMBER *(include area code)* 202-404-7616

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

Table of Diagrams and Figures

RASE - Run-time Automated Schema Evolution

An Implementation of Schema Evolution and Mapping

Purpose

Software needs and requirements change over time as application developers fine-tune and expand the features of a software product. When the software is changed to meet these new requirements, some data structures need to be added and existing ones need to be altered or evolved.

Schema evolution is the changing of preexisting objects and data structures in a program over time. Schema is a definition of the fields contained in an object or structure. A field is a simple data element such as an integer or a string. Schema evolution can include the addition, deletion or changing of fields in a structure. For the purposes of this paper, the term "object" is synonymous with the word "structure" and will be used hereafter.

Schema evolution is needed in relational databases, object databases, and distributed component systems. These systems store objects according to a particular schema or format. Over time, the definition of these objects may evolve. How the schema is allowed to evolve depends on each system's implementation. Some systems may not allow the schema to evolve at all. Others may require the database data and all client programs to be restructured to match the new schema. More sophisticated systems allow the schema to evolve dynamically so the client programs do not have to be recompiled as the schema evolves.

This paper describes an implementation of a sophisticated, dynamically evolving schema. This technique supports evolution of complex object schemas. The schemas definition supports single and multiple inheritance, fixed length arrays, built-in strings, instances of objects and pointers to other objects.

The sample implementation in this paper is geared towards an object database rather than other domains such as a relational database or distributed object system. However, this approach could be used to solve problems in those domains as well. In each of these systems, there is one central program that acts as the server and many possible clients that connect to the server. The server stores or distributes the objects to be used by the client applications. The client applications are often the end user applications.

Background

Any schema evolution implementation has to handle the client side and server side schema evolution. The object database server defines what is capable for schema evolution and the client library defines what the programmer using the system is allowed to do.
Current object database servers usually assume that there is only one true schema for each class. When a change in a class occurs, the corresponding objects in the database must be morphed. The morphing might occur at compile-time, run-time or manually, using database administrative tools. Depending on the system, compile-time systems require a schema maintenance tool to be invoked to update the schema during compilation. Run-time systems delay registering of the new schema until the client program is executed. In either case, the object database server maintains the most recent schema. The database may morph all the object data internally when the new schema is registered or only when the object data is accessed. In a manual system, the user must use either a maintenance tool or send some programming commands to transform the old objects to the new objects. Since the transformation must be done manually, there is large chance for error and corruption but greater flexibility in the range of transformations performed.

Depending on the capabilities of the server, the client code may need to be recompiled as the schema is evolved. In existing object databases, if the server object has been evolved and the client code is using an older version, then the old client program will not be able to access the newly morphed object. In essence, the old client program is considered to be incorrect and the server has the current, correct version.

Current solutions require all client applications to be updated and redistributed whenever the server objects have been evolved. This is a major limitation of the existing schema evolution algorithms.

Manuscript approved December 2, 2001.

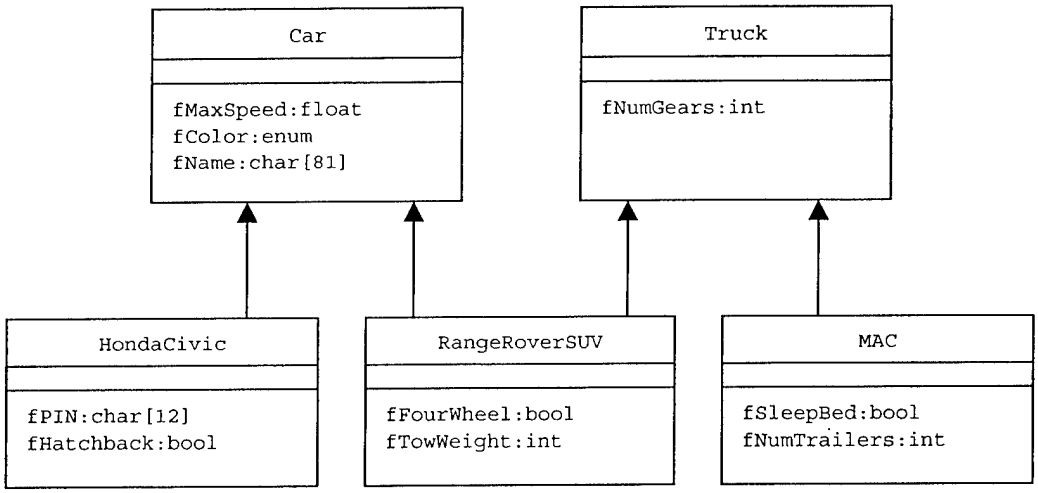
Description and Operation

The schema evolution algorithm being presented in this paper introduces a new solution to the schema evolution problem. The proposing algorithm registers the new schema at run-time but leaves the old schema valid and in tact. This algorithm allows older programs and new programs to coexist. The algorithm is going to be referred to as run-time automated schema evolution or RASE.

Schema Evolution Algorithm Capabilities

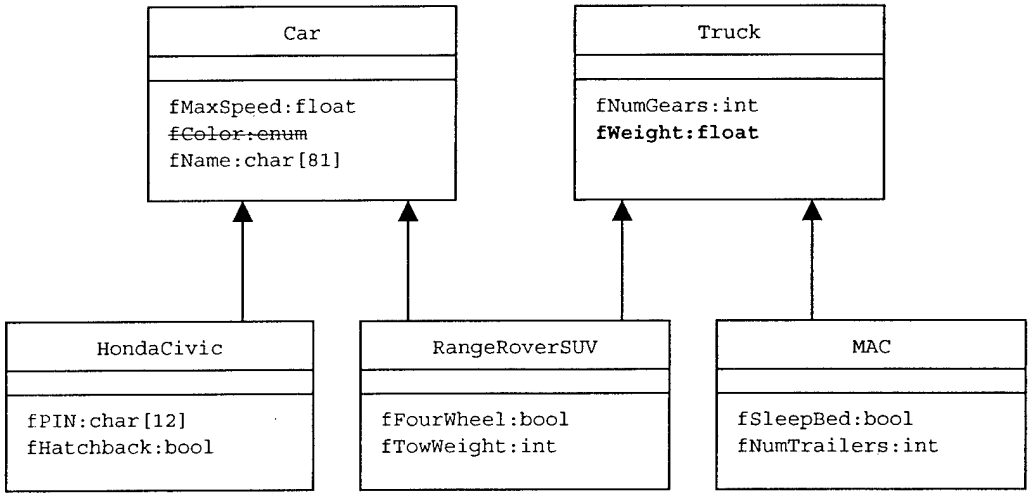
RASE supports several different ways a class can change over time. Any part of a class can change including: adding, removing, or changing the order of a single or multiple inheritance, adding or removing a field, changing a field's type, changing a field's order in the object and renaming a field. The algorithm provides few restrictions about what type of fields can be stored.

The following UML diagrams provides examples of how objects may change over time and what the schema evolution algorithm supports. The classes shown below are shown in their initial state.



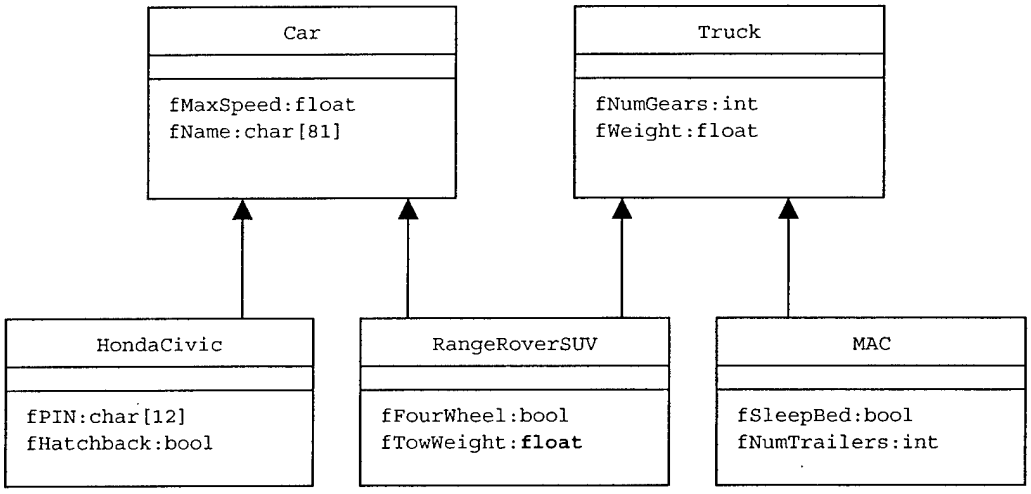
UML Example 1: Initial Schema

Adding and removing fields are one of the most common operations in schema evolution as shown in UML Example 2 on the following page. The field `fColor` is removed from the base class `Car` and `fWeight` is added to the base class `Truck`. Each class has its own corresponding schema so in the given example both class `Car` and class `Truck` schemas would need to be evolved.



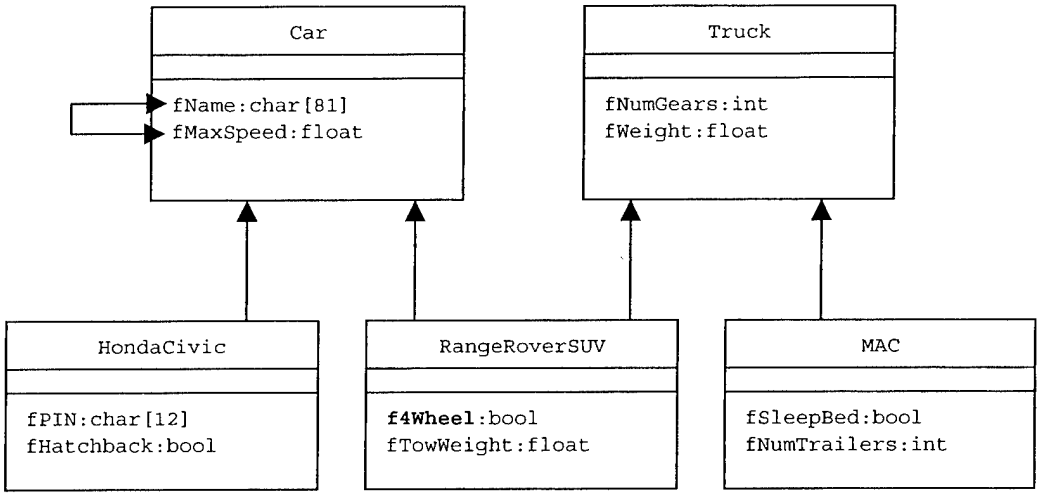
UML Example 2: Adding and Removing Fields from Schema

Another common way schema evolves is to change a type of a field. Shown in UML Example 3, the `fTowWeight` field's type was changed from `int` to `float` for class `RangeRoverSUV`.



UML Example 3: Changing Field Type in Schema

The number and types of variables are only a part of a class's schema definition. The order fields are defined and names associated with individual fields are an integral part of a schema's definition. Merely changing the label or the order in which fields are defined constitutes a schema evolution as shown in UML Example 4 on the next page. In the example, fields for the base class `Car`, `fMaxSpeed` and `fName`, declaration order is reversed and the field `fFourWheel` in class `RangeRoverSUV` is changed to `f4Wheel`.



UML Example 4: Change Order and Label in Schema

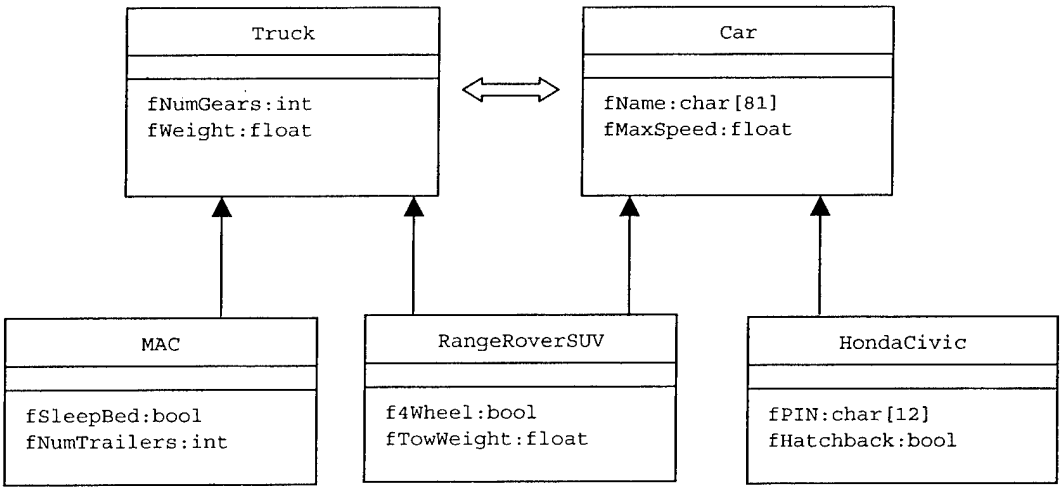
A class's definition is based on its own attributes and the attributes it inherits from parent classes. Adding or removing an inheritance relationship is a schema change. Changing the order of multiple inheritance declaration is also a schema change. Using the current example, the original definition for the `RangeRoverSUV` using C++ is:

```
class RangerRoverSUV : public Car, public Truck
```

A schema change occurs when the declaration order is changed to:

```
class RangerRoverSUV : public Truck, public Car
```

UML Example 5 shown below is a graphical representation of the declaration order change example.



UML Example 5: Changing Inheritance

RASE supports almost any type of field an object can store. The field can be any built in type, a fixed length array, instances of other objects (also known as composition), or pointers to other objects (also known as aggregation). In the proposed implementation of the algorithm, pointers to other objects must use a special templated pointer class that use overloaded arrow operators for pointer de-referencing.

The algorithm does not allow variable length arrays. Variable length arrays are troublesome since there is no way to determine the length of the array at run-time. This means that the user cannot have `char *` as a field but `char[256]` is valid. To handle the most common variable length array, `char *`, a special `String` class is provided for use in the object database. To

handle other variable length arrays, a templated Array class can be provided to provide run-time length information.

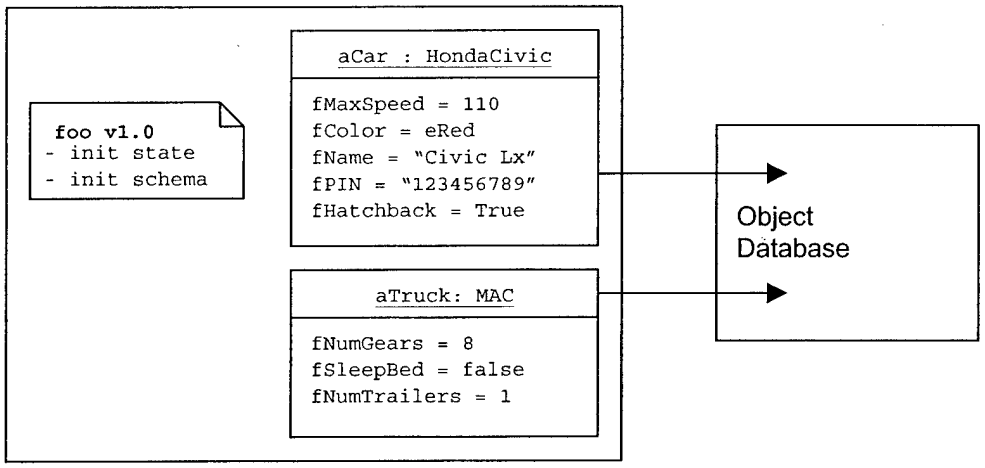
The schema evolution algorithm allows the user to change their classes without forcing limitations on the software design. All normal class changes including any alteration of fields or inheritance have been enumerated in this section and are supported by the proposing algorithm.

Schema Evolution Problem

The schema evolution algorithm needs to allow older client programs and newer client programs to coexist and access the same object data from the server. The schema evolution algorithm also cannot limit how the schema is evolved, meaning the programmer should be able to freely change their classes and the application design.

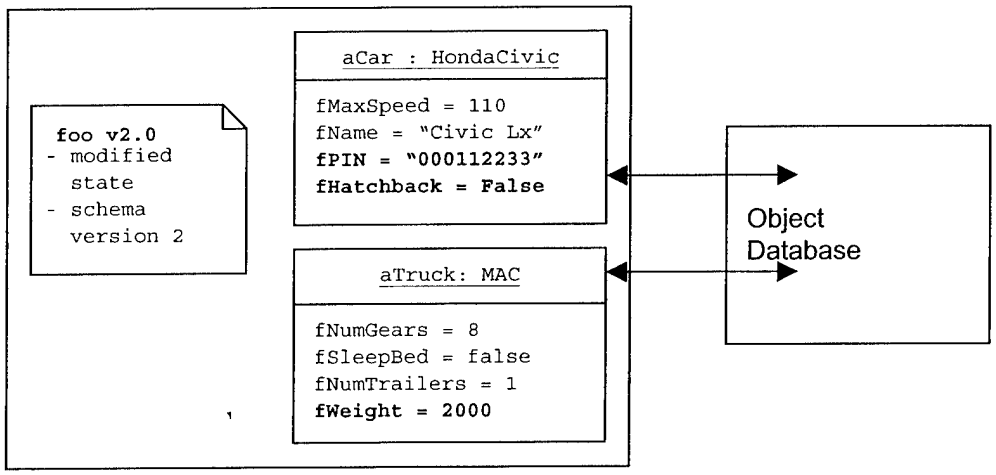
The schema evolution algorithm problem can be broken into two parts. One problem is how the schema and the corresponding objects evolve inside the object database server. The other problem is how the client applications access the server objects regardless of the client schema version. The implementation of the server schema evolution algorithm must support the desired functionality of the client. The following example will describe the desired functionality of the client.

Take a simple schema evolution case where fields are added and removed from classes. The following example will show how the same objects can be accessed by two versions of an application, even though the application versions are using slightly different schema. UML Example 6 shown below represents two objects, `aCar` and `aTruck`, created by client application `foo v1.0`. The two objects are created from the initial schema described in UML Example 1 on page 2. The objects initial state values are stored in the database and shown in the diagram below.



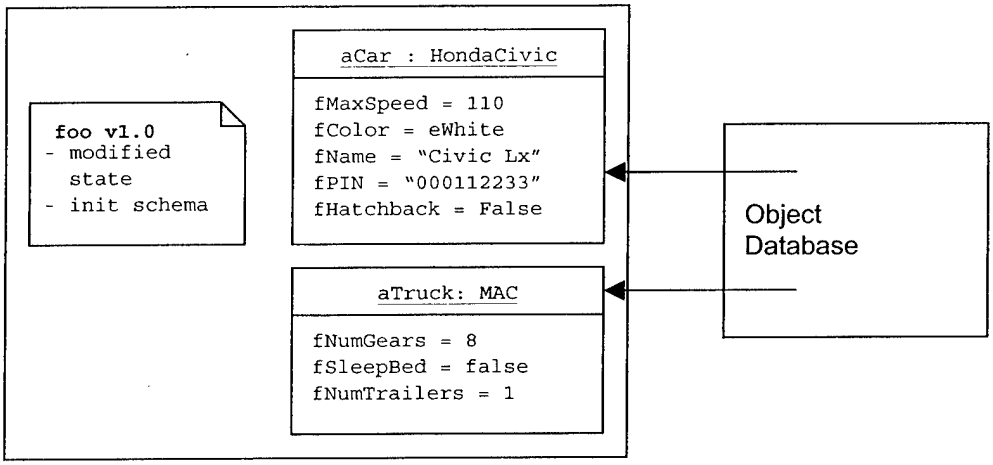
UML Example 6: Objects in Initial State

After the application `foo v1.0` runs, another version of the application, `foo v2.0`, checks out the objects from the database and modifies their values. Except application `foo v2.0` is using the evolved schema described by UML Example 2 on page 3. The schema changes are field `fColor` is removed from `class Car` and `fWeight` is added to `class MAC`. Application `foo v2.0` changes the values stored in fields `fPIN` and `fHatchback` from the object `aCar` and sets a value for `fWeight` in the object `aTruck`. Objects `aCar` and `aTruck` are saved to the database in the schema of v2.0. Any fields specific to the schema of `foo v1.0` are lost. These value modifications can be seen in UML Example 7 on page 6.



UML Example 7: Modified Objects with Evolved Schema

If application `foo v1.0` runs again and checks out the two objects from the object database, it should get the modified changes to `fPIN` and `fHatchback` as shown in UML Example 8 below. Notice that even though version 2 of the application assigned a value to field `fWeight` in object `aTruck`, it was not loaded in version 1 of the application since `fWeight` does not exist in the initial schema. Also notice that the field `fColor` is assigned to a default value, `eWhite`, since the field had been removed from the schema in `foo v2.0`. The proposing algorithm will use an object's default initialization for fields that are added or do not exist in the data from which the object is loading. In this case, the object `aCar` is loaded from data saved in version 2.0 of the schema that no longer has the field `fColor`.



UML Example 8: Retrieving Modified Objects

This example shows how two applications using two schema versions are able to checkout and save the common values of the same objects. This functionality allows for backwards and forward compatibility between different versions of the same application. Although it does not provide optimal support for mismatched schemas, it does allow most evolutions to occur without any difficulty. Having a known, well-defined behavior for mismatched schemas allows developers to determine if a new schema change will affect various components of their systems and take action if needed. The ability for different versions of the same application to coexist is crucial for mature applications where deployment of new application versions is not instantaneous or always necessary.

Schema Definition

All persistent objects need a means to define their schema, as well as an ability to load and save their contents. To provide a consistent means of accessing these functions, it is assumed that every persistent object will implement the `DBObject` interface. In C++, this would be achieved by inheriting from a class called `DBObject`. Figure 1 shows the UML for the `DBObject` class. The `load` and `save` methods are used to serialize the fields of the object into a form that can be transmitted or saved into a file.

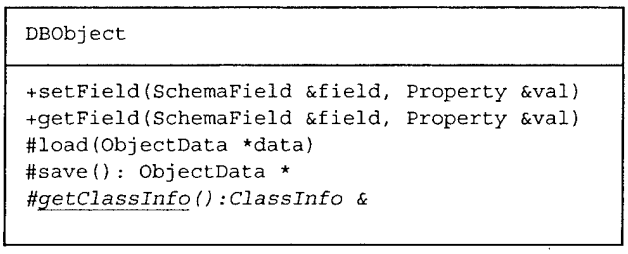


Figure 1: DBObject Interface

The `setField` and `getField` methods provide a means to access the fields of the class. These methods will be used in conjunction with the `getClassInfo` method, which will return the `Schema` and in turn, the fields of the class. The `setField` and `getField` methods use the `SchemaField` class and the `Property` class to facilitate the accessing of the fields. The `Property` class allows for generic storage of any type of field[1].

The `getClassInfo` method is shown in italics because it is not an actual method of the `DBObject` class definition. It is listed here, because each class that inherits from either `DBObject` or an existing persistent object must define a static `getClassInfo` method. The method is static, so that a user can obtain schema information about a particular class without having to actually create an instance of the class. The details about the `ClassInfo` class are shown in Figure 2 on page 8. Since the `getClassInfo` method is static, there is one, and only one instance of `ClassInfo` for each inheritance level of a persistent class. This also means that there is only one instance of a `Schema` class for each level of inheritance, since `ClassInfo` is composed of a `Schema`.

The `ClassInfo` class provides a single place to find information about a particular persistent class. The user can determine the name of the class, and obtain a reference to the `Schema` for that class. The `Schema` class also allows the user to determine the class name, in addition to the super classes and fields of the class. `Schema` inherits from a class called `CommonSchema`. The `CommonSchema` class represents the elements of a schema that are common to both the client and server parts of the system. The `Schema` class adds elements that are specific to the client program. Specifically, methods have been added to determine if the schema has been registered with the server. The Schema class also adds a `SchemaMap` pointer that will be used to load a particular instance of the class the schema represents.

[1] The design of the Property class and its ability to perform generic storage and field type conversion is described at length in the Naval Research Lab technical memo: *A Generic Preference System Pattern and C++ Implementation.* NRL/MR/5707-00-8473, September 29, 2000.

The fSuperClass field in CommonSchema is a list of pointers to the super class or classes of a particular schema. The order in which super classes are added to the list determines the order of inheritance. CommonSchema is composed of a list of schema fields, represented by the SchemaField class. This class allows specification of the field name and type. It also keeps

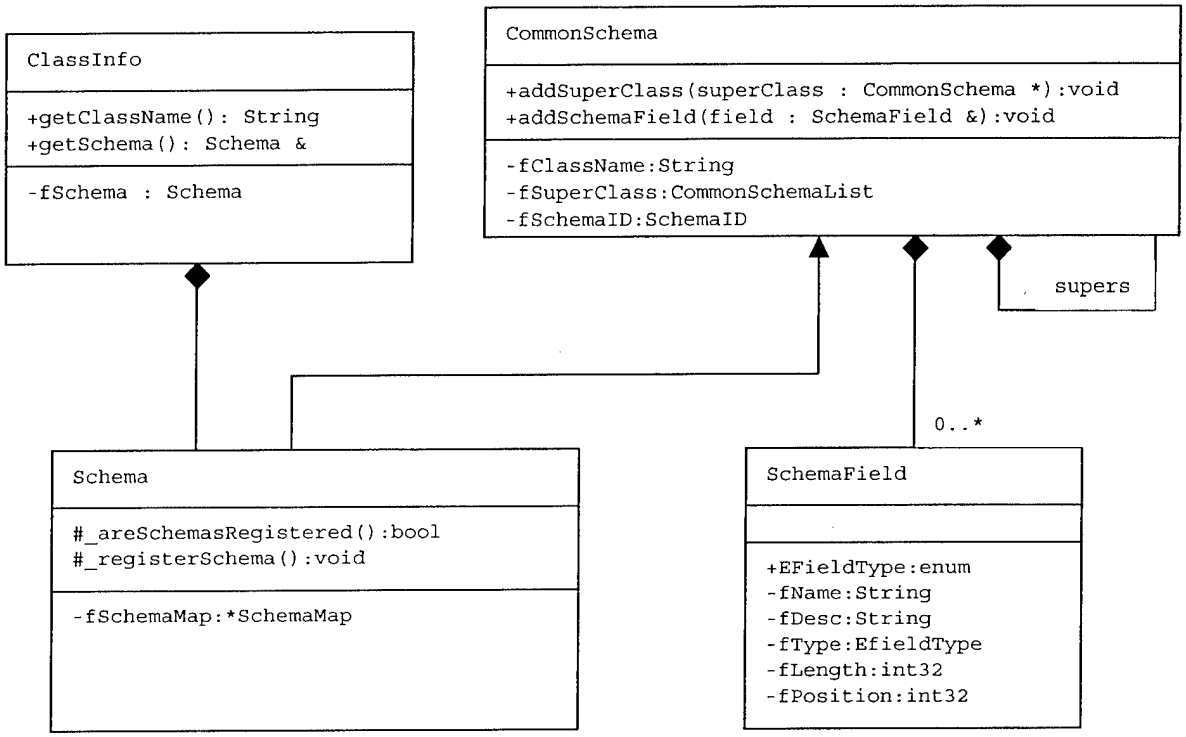


Figure 2: Schema and ClassInfo

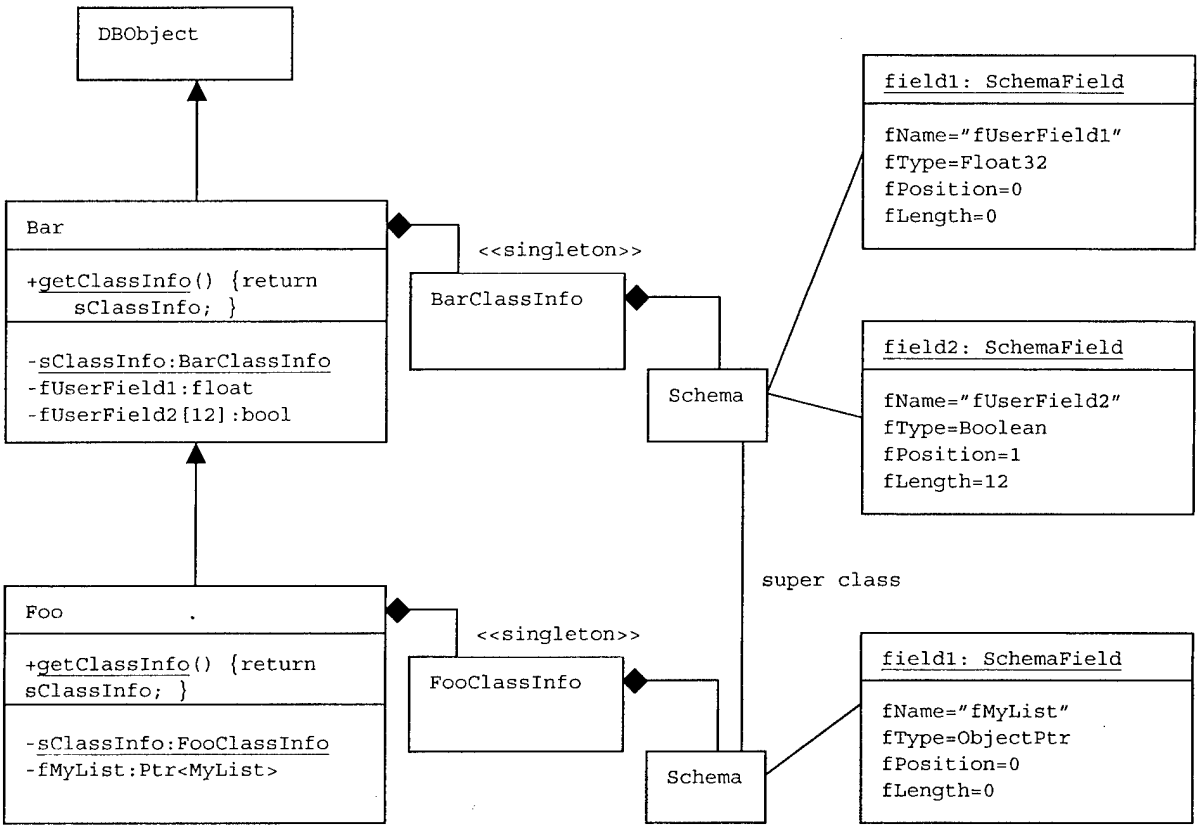
track of the order of the fields in the Schema using the fPosition field. This field indicates the position number of the field in the Schema. Position numbers start at zero for the first field and go to n-1 for the last field, where n is the number of fields in the schema. The fType field is an enumeration that indicates the data type of the field. This enumeration includes the following types: eUndefined, eEnum, eBoolean, eChar, eInt8, eUint8, eInt16, eUint16, eInt32, eUint32, eFloat32, eDouble64, eString, eArray, eDBObjectInstance, eDBObjectPtr. The fLength field is used to represent fixed length arrays. A length of 0 indicates a single instance of the type specified in the fType field. A length greater then zero indicates a fixed length array of the specified type. A length of –1 indicates an unknown or uninitialized length. Since strings are used so frequently, but are variable length, we have included string as a built-in type. The Array type indicates all other variable length arrays, which is an instance of a special template array class. The DBObjectInstance type indicates instances of other persistent objects. The DBObjectPtr type specifies pointers or references to other persistent objects.

For each persistent class that is declared, an implementation of the setField, getField, and getClassInfo methods must be provided. In a real object database system, a code generator is used to convert the object definitions, in IDL or ODL, into the particular language the user wants[2]. The code generator creates a specific subclass of ClassInfo for each level of

[2] IDL (Interface Definition Language) and ODL (Object Definition Language) are standard languages for defining object schema and interfaces of objects. The Object Management Group (OMG) defines the IDL and ODL standards.

inheritance the user defines. The constructor of each subclass of `ClassInfo` creates and adds the `SchemaField` objects to the instance of `Schema` contained in `ClassInfo`. A static instance of the specific `ClassInfo` class is declared inside each level of inheritance for the user's classes. The static `getClassInfo` method returns a reference to the static subclass of `ClassInfo`. The code generator also creates the header file and implementation for each user declared persistent class. For each class it would provide an implementation of the `load`, `save`, `setField`, and `getField` methods, along with various other methods necessary to provide the functionality the database supports.

UML Example 9 shows an example of a `class Foo` that inherits from `Bar`, that inherits from `DBObject`. Since `Foo` and `Bar` are both independent levels of inheritance from `DBObject`, they both must implement the `getClassInfo` method. A specific subclass of `ClassInfo` is created



UML Example 9: Schema Example

for both the Foo and Bar classes, called FooClassInfo and BarClassInfo. The constructors for these methods create and add instances of the SchemaField class to their Schema. Code Example 1 shows an example implementation of the BarClassInfo and FooClassInfo constructors.

```
BarClassInfo::BarClassInfo()
    {
    setClassName("Bar");
    getSchema().addSchemaField( SchemaField(getSchema(), "fUserField1",
                                "User Field 1", SchemaField::eFloat32, 0, 0));

    getSchema().addSchemaField( SchemaField(getSchema(), "fUserField2",
                                "User Field 2", SchemaField::eBoolean, 1, 12));
    }

FooClassInfo::FooClassInfo()
    {
    setClassName("Foo");
    getSchema().addSuperClass( &Bar::getClassInfo().getSchema());
    getSchema().addSchemaField( SchemaField(getSchema(), "fMyList",
                            "Pointer to MyList", SchemaField::eDBObjectPtr, 0, 0);
    }
```

Code Example 1: FooClassInfo and BarClassInfo Constructors

In general, there are three steps to constructing the ClassInfo subclass. First, set the class name using the setClassName method. Next add a pointer to any super classes. Finally, call addSchemaField for each persistent field in the class. The addSchemaField method is relying on the copy constructor for SchemaField to create it's own copy of the SchemaField that is passed in. Because of this, a temporary local instance of a SchemaField can be passed in to the addSchemaField method. The addSuperClass method, in contrast, maintains a pointer to the actual super class schema that is passed in, allowing the user to easily traverse up the chain of inheritance and access the real super class's schema.

Schema Mapping Algorithm

The schema mapping algorithm can be broken into two parts, the client side and the server side. The client must load and save user objects, and register user schemas with the server. The server must store current schemas and their schema ids, allocate new schema ids, and create schema maps. The following rules and assumptions will be made about the schema-mapping algorithm described in this paper. Each client contains it's own schema definition for all persistent classes they wish to exchange with the server. Each unique schema will be assigned a unique schema id by the server. The server will maintain a list of registered schemas and their associated id's. Before client programs can exchange objects with the server, they must register their schemas with the server. The server will match the schemas against its list of existing schemas and return the appropriate ids. If a schema does not match any existing registered schema, it will assign it a new schema id.

When saving an object to the server, a client program must serialize the contents of the user's object into an instance of an ObjectData class. The users object, along with its schema id, is saved in the form of its own schema. In the future when serving ObjectData to a client, the server will deliver the ObjectData in the schema of the client that last saved the information. The loading algorithm for the client objects assumes that all ObjectData is saved in a different variation of its schema and needs to be mapped. Upon request, the server can provide the client with a SchemaMap that maps a particular schema id to all other variations of the class the schema represents.

To implement object loading and saving, and schema mapping, four classes are needed.

- ObjectData
- FieldData

- SchemaMap
- FieldMap

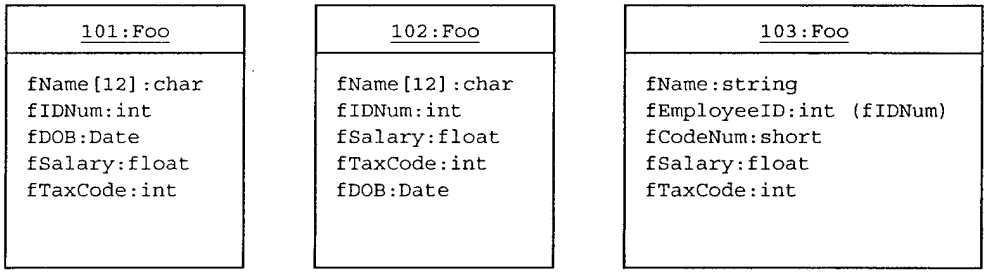
The ObjectData class represents the contents of a persistent object at one level of inheritance. It contains the object's schema id, an ordered array of FieldData objects, a list of ObjectData's for fields that are instances of other objects, and a list of pointers to the ObjectData for any super classes. The FieldData contains the field type information along with the actual data for the field. The ObjectData class is used to send and receive the object's contents to the server. The SchemaMap class is used to represent the mapping of one particular schema to all other variations of that same schema. The SchemaMap class contains a list of FieldMap objects, one for each variation of the schema. These FieldMap objects are indexed according to the id of the schema they represent. The FieldMap contains an array of schema field position numbers. The array has one entry for each field in the client schema. The position numbers indicate which ObjectData field is loaded into the client field.

The client side algorithm consists of four steps.

1. **Initialize the client schemas.** At startup time, the client program creates the schemas for each persistent user class. The schema id for each of these schemas is set to zero, indicating that the schema hasn't been registered with the server yet.
2. **Obtain schema ids from the server.** Before any object data can be transferred between the client and the server, the client's schemas need to be registered with the server. Once a schema is sent to the server, the server looks to see if the schema has been registered previously. If the schema has been registered then the previous schema id is returned, otherwise a new schema id is assigned.
3. **Obtain the SchemaMap from the server.** Once a schema id for a particular schema has been obtained from the server, the client can request a current schema map for that schema id at any time. The schema map shows how to map the client schema into other variations of that class. Figure 3 and UML Example 10 show this for a class called Foo. The client has one version of Foo with an id of 101. The server knows of

Field #	Schema id 101	Schema id 102	Schema id 103
0	0	0	0
1	1	1	1
2	2	4	-
3	3	2	3
4	4	3	4

Figure 3: SchemaMap Example



UML Example 10: SchemaMap Example

two other variations of Foo and has assigned them id's 102 and 103. The UML notation for the three versions of Foo in UML Example 10 shows how each version differs from the other. Figure 3 shows what the contents of the SchemaMap would look like if the client with schema id 101 requested a schema map from the server.

4. **Load or save contents of client objects.**
    a. When saving an object from the client to the server, it is not necessary to have the schema map for the object, since the schema map is only used to map fields of different schemas to the client's schema. When saving an object, it should always be saved in the schema of the client who is saving it. Saving an object is simply a matter of calling the virtual save method for the persistent object. This method will create an `ObjectData` instance and fill it with the contents of the object. It will also take care of saving the data for any super classes. The `ObjectData` is then returned from the save method and sent to the server to be stored.
    b. To load the contents of an object, an instance of `ObjectData` must be sent from the server to the client. This `ObjectData` is then passed to the virtual load method in the object being loaded. The load method looks up the schema id in the `ObjectData` and uses this id to get a `FieldMap` from the `SchemaMap` for the class being loaded. Each field stored in the `ObjectData` is then mapped to a corresponding field in the object being loaded using the `FieldMap`.
    c. As each field is loaded, the type of the field indicated by the `FieldData` in the `ObjectData` is compared against the type in the `SchemaField` in the `Schema` for the object. If the two types match, then a binary load of the field can be done. If the field types don't match, then a type conversion will have to be done. This is done using the `Property` library, which allows a `Property` class to represent an instance of any type. Additionally it allows one type to be converted to another type. The Property library is implemented using templates and run-time type identification. Because of this, it is very flexible, but does not provide the highest possible performance. Since the `Property` library is only used if the types of a field don't match, the user should see optimal performance during normal use of the schema-mapping algorithm.
    d. The actual data for each field is stored in the `FieldData` class inside the `ObjectData` class. The only exception to this is the storage of instances of other persistent objects. The `ObjectData` contains a list of `ObjectData`s for fields that are instances. If a field's type indicates that it is an instance of another object, then the `ObjectData` for that field will have to be found in the instance list and then passed to the load method of the object instance.

In practice, the order in which events occur is different then they have been stated so far. The following list shows a realistic sequence of events that would occur when saving an object.

1. The user creates a new object and tries to save it to the server.
2. Client code checks for a valid schema id and sees that it hasn't been set yet.
3. Client sends the schema to the server and the server replies with the id for the schema along with the ids for all of its super classes.
4. The user's object is encoded into an instance of an `ObjectData` class using the virtual save method.
5. The `ObjectData` is sent to the server to be saved.

The following list shows an example of the order of events when trying to load an object from the server. In this example, we are assuming that this is the first time an object of this class has been loaded from the server.

1. The client requests the loading of a particular object from the server.
2. The server returns the `ObjectData` for the client's object.
3. The `ObjectData` is passed into the virtual load method of the client's object.
4. The load method tries to get the `SchemaMap` for the object from its schema. The `SchemaMap` doesn't exist yet.

5. The load method needs the schema id in order to request the `SchemaMap` from the server. The schema hasn't been registered yet, so the schema doesn't have an id yet.
6. Client code sends the schema to the server and receives back the schema id of the class.
7. Client then requests a `SchemaMap` from the server using the schema id. Client receives back the `SchemaMap`. The `SchemaMap` is stored in the Schema for later reuse.
8. The load method loads the contents of the object using the `ObjectData` and the `SchemaMap`.

There is a special case that needs to be addressed when loading objects. This is when a different client saves an object with a new variation on a schema we have already loaded. In this case, the client program has an existing `SchemaMap` that doesn't contain an entry for the new schema stored on the server. If the client were to try to load this object, it would need to deal with this situation. The following list shows the steps for handling this case.

1. User requests the load of an object recently saved to the server by a different client with a different schema.
2. The server returns the `ObjectData` to the client.
3. The `ObjectData` is passed in to the virtual load method of the user object.
4. The load method finds the existing `SchemaMap` and tries to get the `FieldMap` for the schema id contained in the `ObjectData`.
5. The `FieldMap` is not found, at this point it is determined that the `SchemaMap` must be out of date and a new `SchemaMap` should be requested from the server.
6. The client code requests a new schema map from the server using the client's schema id for the object that is being loaded. The server returns a new `SchemaMap` for the requested schema id.
7. The load method tries to find the `FieldMap` again, this time it succeeds.
8. The user object is loaded using the `FieldMap`.

The server has three primary responsibilities. They are saving registered schemas, allocating new schema ids, and creating schema maps. The tasks of registering schemas and allocating new schema ids are closely related, since the client never explicitly requests either. The client simply requests the schema id for a schema. The client doesn't know if the schema is already registered, or is a new schema. The following list shows the steps used to register a new schema on the server.

1. The server receives a request for the schema ids of a class and its super classes.
2. The server gets a list of all previously registered classes with the same class name.
3. It then searches the list for a schema that matches the schema for which the client has requested an id. A schema is considered a match if it inherits from the same classes, has the same number and order of fields, and has the same names and types. Inheritance is matched based on class name, not schema id.
4. Once it determines that there is no match for the client's schema, the server allocates a new schema id and saves the schema in its list of registered schemas.
5. At this point a recursive call is made to get the schema id for any super classes.
6. A list of class names and schema id pairs is returned to the client.

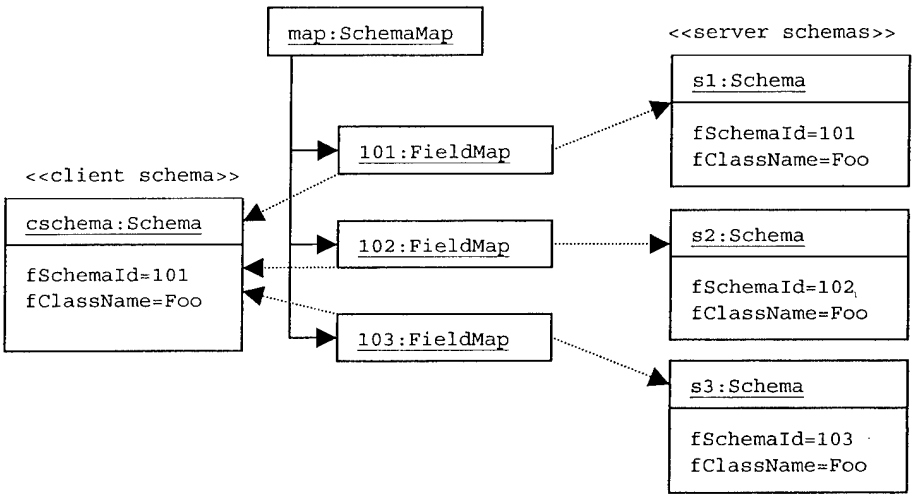
The next list shows the steps involved with providing the schema id for a class that has already been registered with the server.

1. The server receives a schema along with a request for its id from the client.
2. The server gets the list of all schemas that match the class name of the client's schema.
3. It then looks for a matching schema in the list, and an exact match is found.
4. The schema id of any super classes is obtained by recursively requesting their ids

5. A list of class names and schema id pairs is returned to the client.

When a client request a schema map for a particular schema, the client needs to only provide the schema id of the schema for which it needs a map. The server must first get the schema of the client's class using the id, and then build a `SchemaMap` for it. For the purposes of this discussion, the client schema will be the schema from which we are mapping, and the server schema will be some variation of that schema to which we are mapping.

1. The server receives a request from a client for the schema map of a particular schema id.
2. The server gets the schema for the specified client schema id.
3. Next it gets a list of all schemas that have the same class name as the client schema. This list should contain an entry for the schema that the client is requesting the map for.
4. Next, the server has to build the `SchemaMap` by going through the list of schemas and creating a `FieldMap` that maps from the schema in the list to the client schema.
    a. First it creates a new `FieldMap` with the number of entries set to match the number of fields in the client schema.
    b. Next it must go through all the fields in the client schema and find a matching field in the server schema based on field name or previous field name.
    c. If a match is found, then the server's field number is entered in the `FieldMap` at the position indicated by the client's field number.
    d. If no match is found, then a value of –1 is entered in the `FieldMap` for that client field. This indicates that there is no match for the client's field in the server's schema.
5. All the resulting field maps are accumulated into an instance of `SchemaMap`. UML Example 11 shows what the resulting `SchemaMap` would look like.
6. The `SchemaMap` is returned to the client.



UML Example 11: FieldMap Object State Diagram

Client Schema Mapping Implementation

Now that the algorithms for schema mapping have been described, the exact implementation and interaction of the classes `ObjectData`, `FieldData`, `SchemaMap` and `FieldMap` will be explained.

The SchemaMap class uses FieldMap objects to map fields from a client schema to each schema variant. The UML class diagram for SchemaMap and FieldMap is shown in Figure 4 below.

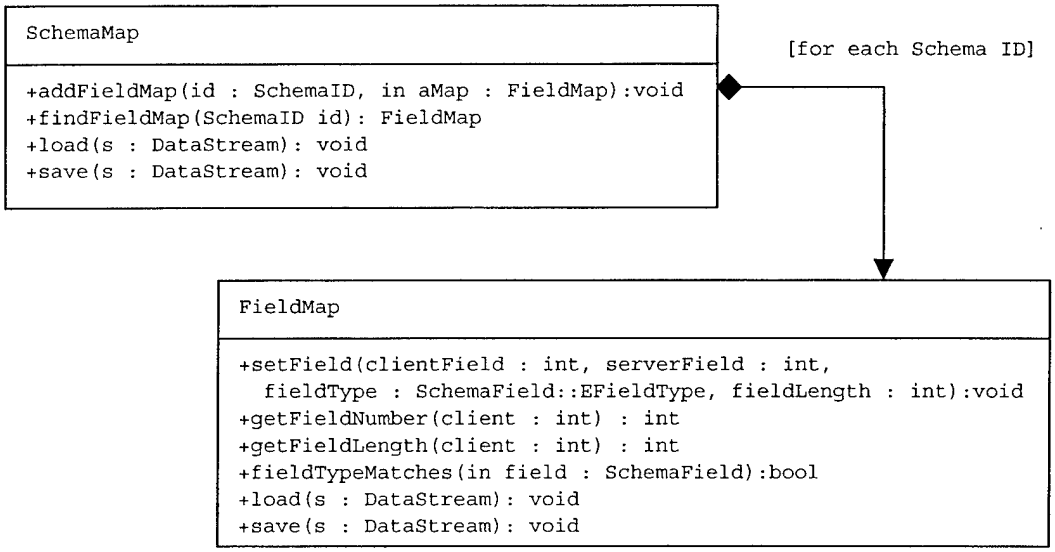


Figure 4: SchemaMap and FieldMap

SchemaMap maps a schema ID to a FieldMap. The SchemaMap should contain all possible schema IDs and hence all possible FieldMap objects. In practice, a SchemaID can be a long int or a class composed of multiple ids such as database id and a unique id within the database. If a schema ID is not found, then a new schema has probably been registered on the server and a new SchemaMap needs to be requested. The FieldMap maps the client's Schema fields to the fields in another version of same class. FieldMap's accessor methods getFieldNumber, getFieldLength and fieldTypeMatches take the client field number and return the appropriate information for the field it maps to in a server schema. SchemaMap and FieldMap will be created on the server side and used on the client side. Both objects contain load and save methods for serialization that allow them to be passed from the server to the client easily. DataStream is used by load and save to represent a binary stream that operates on some device. A possible device can be a file, socket, or buffer.

SchemaMap and FieldMap are used in the loading and saving of DBObject objects and their associated data. ObjectData and FieldData are classes that store the fields' data and provide random accessing of the fields' data. The UML Diagram for ObjectData and FieldData is shown in Figure 5 on page 16.

Each object's data is actually stored in the FieldData class. The FieldData class stores the binary data for each field of an object. There are four broad types of data that can be stored: built in types, arrays of built in types, object instances and arrays of object instances. Built in types are the compiler-supported types such as bool, int, float, double. The methods addField and addFixedArrayField must be overloaded to support all possible types of fields to be stored, including basic types and special database primitives. The actual implementation of each basic type may vary with the programming language. In C++, templated member functions allow one function to be overloaded once for each type. In Java, the class would require each method to be written for every type. The method addInstanceField and addInstanceFieldList are used to allow composite relationships with a child class or classes. Object instances are objects that are not pointers to other objects but actual instances of another object within one object. The addInstanceField adds a single object instance and addInstanceFieldList adds an array of object instances.

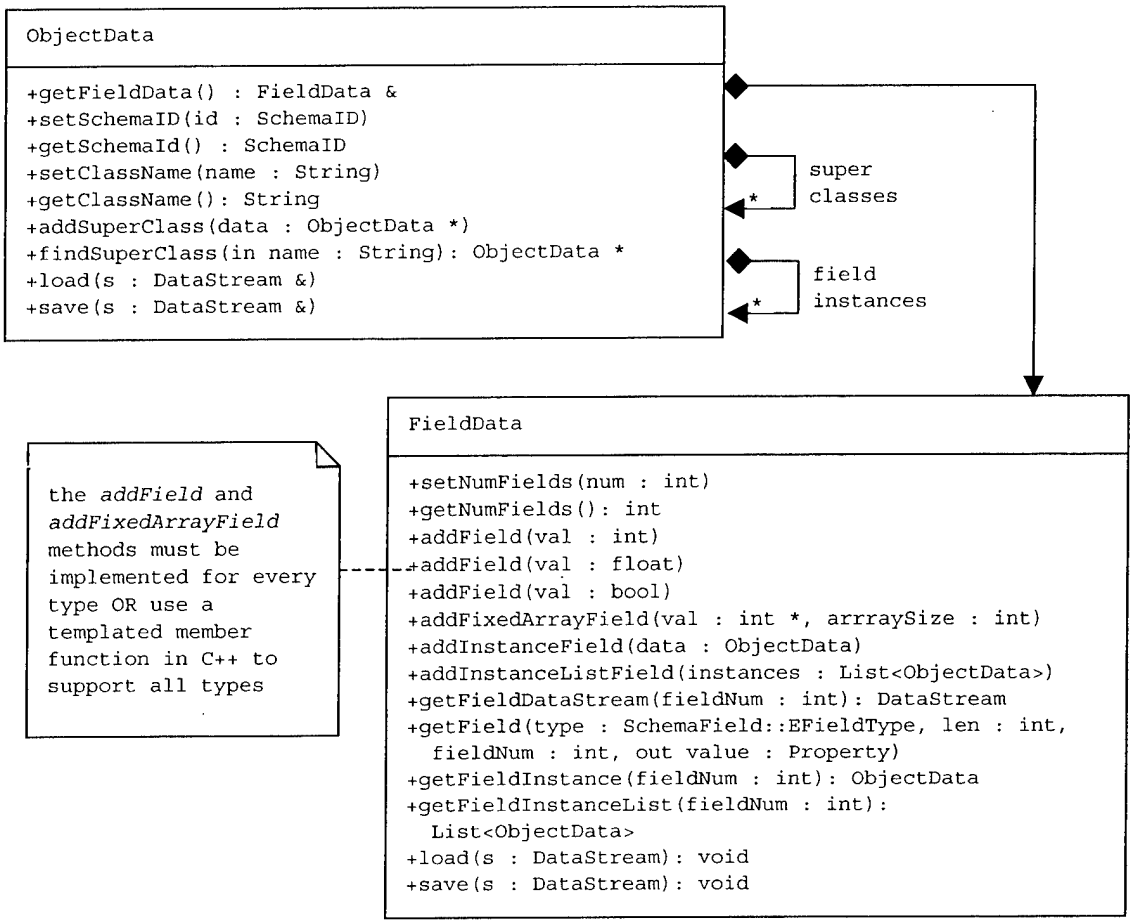


Figure 5: ObjectData and FieldData

Implementations may vary for how the binary data should be stored inside `FieldData`. In our implementation, the `FieldData` class uses a byte array to store the raw field data and an index array to give the position where each field starts in the byte array. Whatever the implementation, field data needs to be accessed randomly based on its position to support the loading process. `ObjectData` and `FieldData` serialization methods `load` and `save` allow the objects to be passed back and forth between the client and the server.

There are multiple ways to extract data back out of `FieldData`. The method `getFieldDataStream` provides raw access to a field and should be used when there is no type mismatch for a basic type. The method `getField` returns the data in a `Property` and is generally used for type mismatch cases. The `getFieldDataStream` and `getField` methods provide access to data stored using any of the `addField` or `addFixedArrayField` methods. The method `getFieldInstance` returns the `ObjectData` for a field number. This method is used to extract the data that is stored by `addInstanceField`. Likewise, the method `getFieldInstanceList` returns the `ObjectData` list stored by `addInstanceFieldList`.

The algorithm that handles the client side loading and saving is encapsulated into a class called `DBSerializer`. The `DBSerializer` class is shown in Figure 6 on page 17. `DBSerializer` is a base class that provides three common methods and three pure virtual methods that need to be implemented by each subclass. There must be a `DBSerializer` subclass for every level in a `DBObject` subclass and therefore for every schema. The `DBSerializer` constructor takes a reference to a `DBObject` that it is serializing. The methods

load, loadFields and save are implemented once in the DBSerializer base class. The methods setFieldData, saveFields and getSchema must be implemented for every subclass. The implementations of the methods are described in the following paragraphs and sequence diagrams.

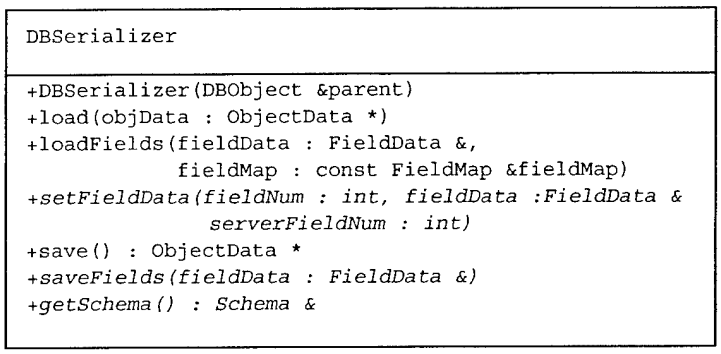


Figure 6: DBSerializer Class Diagram

The implementation of saving objects in the client involves the DBObject, DBSerializer, ObjectData, FieldData and Schema classes. The sequence diagram for the client-side saving algorithm can be seen in Figure 7 on page 18. The diagram shows how a simple class DBFoo is saved to the database in a client application. Class DBFoo in this example simply inherits from DBObject and has two fields, field1_str and field2_int.

The sequence diagram shows the schema registering process and the serialization of a simple object. The diagram starts with a user invoking the save method. Subclass implementations of the DBObject save method use an associated subclass of DBSerializer to serialize the object into an ObjectData. The first thing the DBSerializer does is to ensure the Schema associated with the DBFoo has been registered. The schema is retrieved at the DBFooSerializer level since the subclass knows about the static DBFoo::getClassInfo method. If not already registered, the Schema can than register itself with the server and receive a valid SchemaID back from the server. In the sequence diagram, the ServerStub is a class that manages all socket communication with the database server process. Once a valid SchemaID has been assigned to DBFoo's schema, the DBSerializer can create and fill in the ObjectData. The SchemaID and class name are stored inside the ObjectData. The saveFields method is then called passing it the ObjectData's FieldData. The saveFields method must be implemented at the DBFooSerializer level since only the subclasses know what fields should be saved for each object. DBFoo's two attributes are stored by calling FieldData's addField method, passing the current value for each field. In this simple case, DBFoo only contains two variables and no super classes so the serialization process is finished. The ObjectData is then sent to the server via the ServerStub class and the ObjectData is destroyed on the client side. In more complex cases where saving objects requires multiple levels of inheritance, each DBObject subclass's save method would invoke the super's save method and store the super class ObjectData inside the subclass's ObjectData by calling ObjectData's addSuperClass method.

The next sequence diagram on page 19 describes how to load an object.

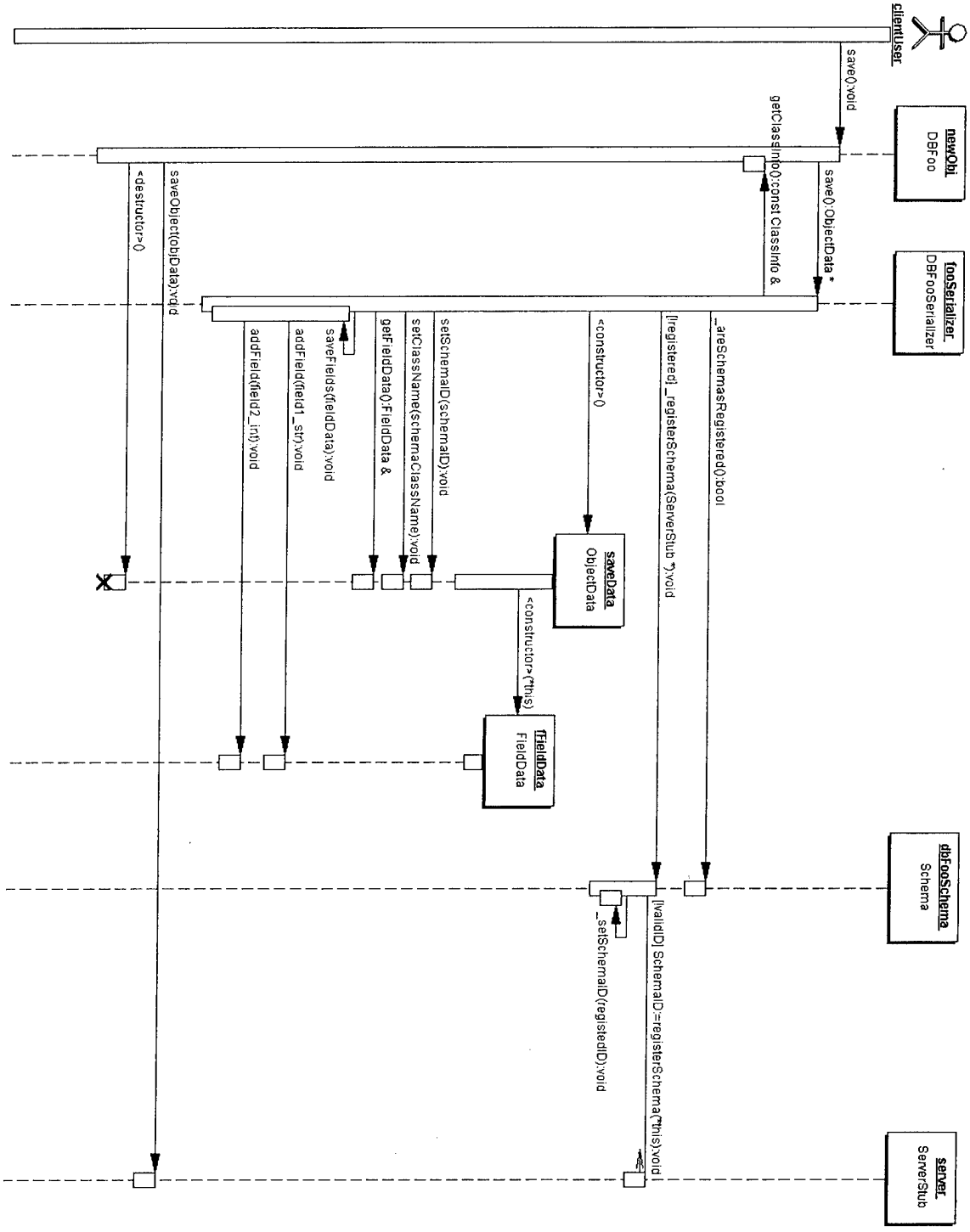


Figure 7: Saving DBObject Sequence Diagram

The implementation of the client-side loading algorithm for a subclass of `DBObject` is shown through the interaction of `DBSerializer`, `ObjectData`, `Schema`, `SchemaField`, `FieldMap`

and `FieldData` classes. The sequence diagram showing the algorithm and the classes interaction is shown below in Figure 8. The sequence diagram describes loading the same simple `DBFoo` object that was used for the saving diagram in Figure 7 on the previous page. The sequence diagram below assumes the `ObjectData` has already been retrieved from the server and all schemas are already registered.

The loading algorithm is initiated by calling `DBSerializer`'s `load` method with the `ObjectData` that was retrieved from the server as the sole parameter. The `load` method starts by retrieving the `FieldMap` from `DBFoo`'s `Schema`. The `Schema` retrieves the `FieldMap` using the `ObjectData`'s `SchemaID` to find the appropriate `FieldMap` via the `SchemaMap`. The

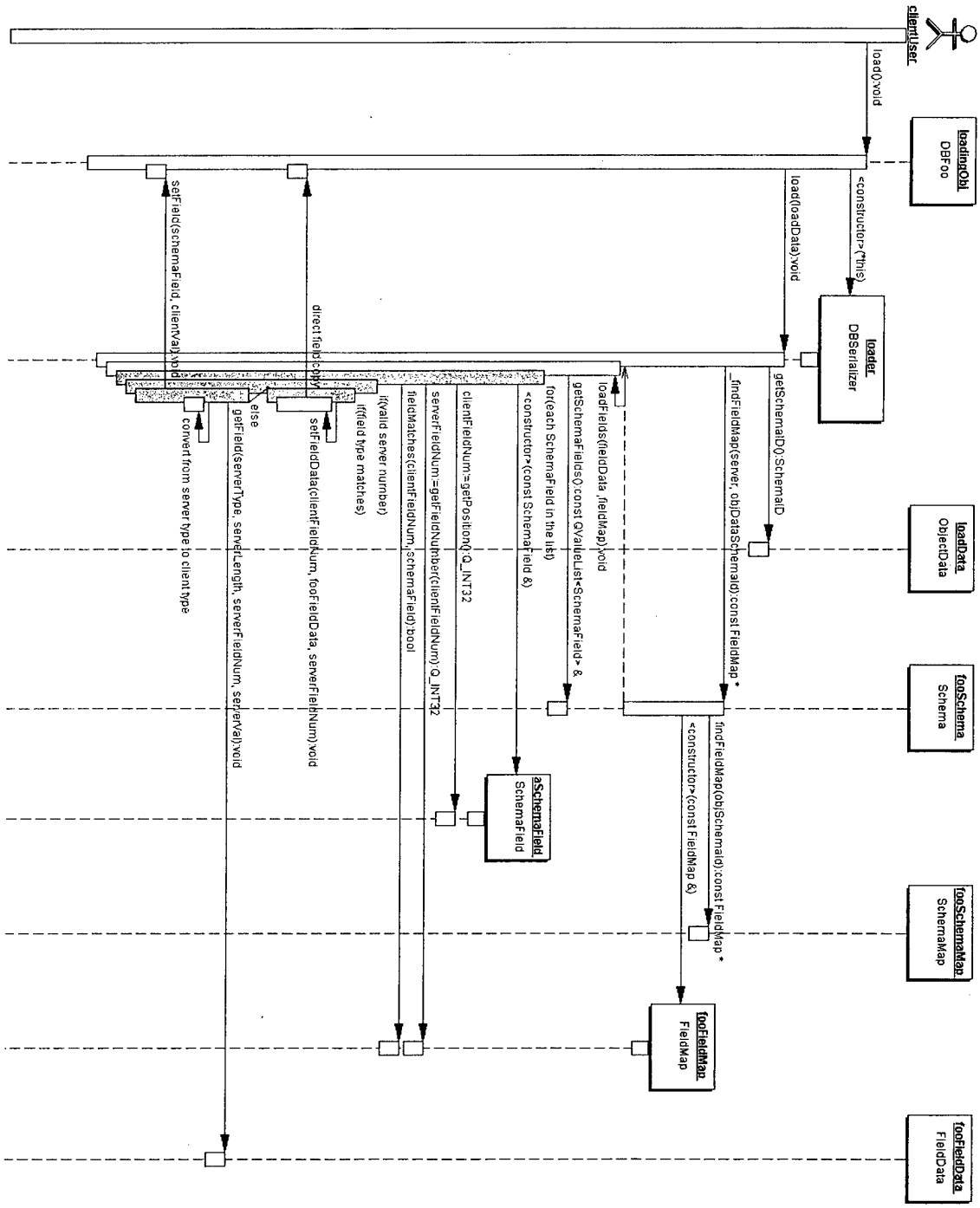


Figure 8: Loading DBObject Sequence Diagram

ObjectData's SchemaID is used to lookup the FieldMap. Since the ObjectData originates from the server, the loading algorithm needs to map the data from the server schema to the client's schema. The DBSerializer's loadFields method is then called with the FieldData and the FieldMap as arguments. The FieldData is retrieved from the ObjectData.

The loadFields method loads the FieldData by using the associated FieldMap. The FieldData associated schema may be a different version from the client's schema so the FieldMap must be used. The algorithm starts by iterating through the client's SchemaField objects. Inside the for loop, the initial task is to use the client field number to get the associated server field number from the FieldMap. Calling the FieldMap's getFieldMap method, passing in the client field number, retrieves the server field number. The client field number and server field number are defined by their order in the schema. If there is no associated server field, then the server number will be invalid and the client field's value will remain in its default state and the loop will continue. Otherwise, the client field has an associated server field and it will be loaded. The direct field copy, implemented in DBFooSerializer's setFieldMethod, uses the FieldData's getFieldDataStream method to extract and assign the value. If the client field and the server field are of the same type, then a direct field copy can be done. If the client field and the server field are of different types, then a type conversion must be done. In this implementation, the Property converters are used to convert a value from one type to another. For the type mismatch case, the DBFoo class method setField is used to set the field's value. Note that two separate methods are used for the direct field copy versus the type mismatch case. The direct field copy method is necessary since it is more efficient than the type mismatch method.

Server Schema Management Implementation

The server is responsible for keeping track of registered schemas, allocating new schema ids, and creating schema maps. The means by which the server stores registered schemas is beyond the scope of this paper, since it would require too much knowledge of the purpose of the system that is using these schema algorithms. For the purposes of discussing the implementation of the schema id allocation and schema map generation algorithms, it is assumed that a class SchemaFile exists. A single instance of the SchemaFile class exists in the server. The UML for the SchemaFile class is shown in Figure 9. The SchemaFile class stores all registered schemas and allows the server to get a particular schema from a schema id, and to get a list of

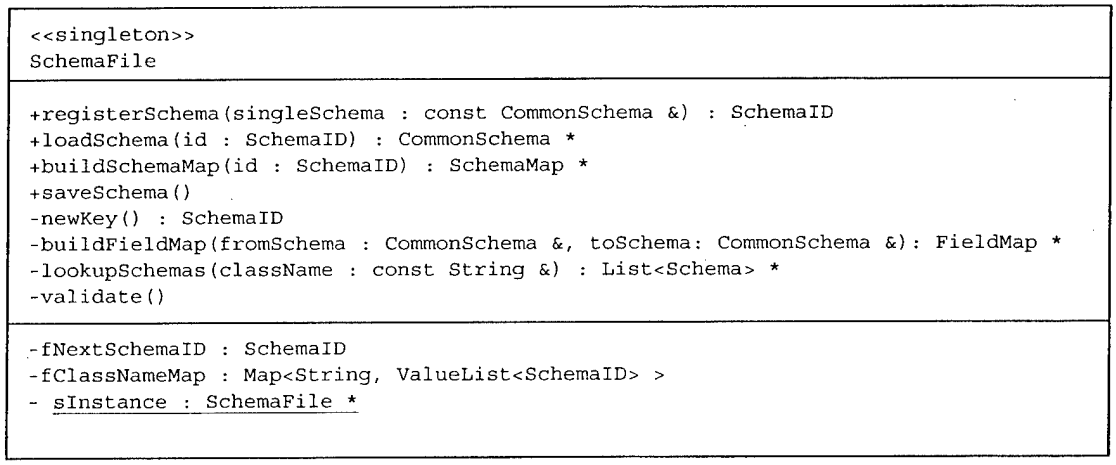


Figure 9: SchemaFile Class Diagram

schemas from a class name. The methods for actually saving and loading a schema are not shown in the UML diagram and will not be discussed. The important methods in SchemaFile are the registerSchema and buildSchemaMap methods. The C++ implementation of these methods is well documented in Code Example 2 below. The SchemaFile class contains two

important fields. One is the `fNextSchemaID` field, which is an instance of `SchemaID`. In this sample implementation, `SchemaID` is just a `typedef` for an integer. The `newKey` method will be used to allocate new `SchemaID's` by incrementing the `fNextSchemaID` field and returning it's current value. The `fClassNameMap` field is used to keep track of all schemas that have the same class name. It does this by associating a class name with a list of `SchemaID's`. As schemas are registered, the class name map is updated to include the new schema.

Whenever requests to register a schema or get a schema map are received in the server from a client application, a handler class will be created to handle that particular client request. Figure 10 shows the UML for two classes called `HandlerSchemaID` and `HandlerSchemaMap`. Each of these classes have an `execute` method that gets called to perform the actual processing of the client's request. The `execute` methods will use the `SchemaFile` singleton to perform the task and then send a reply back to the client. The implementation of receiving the request and sending the reply to the client will not be discussed.

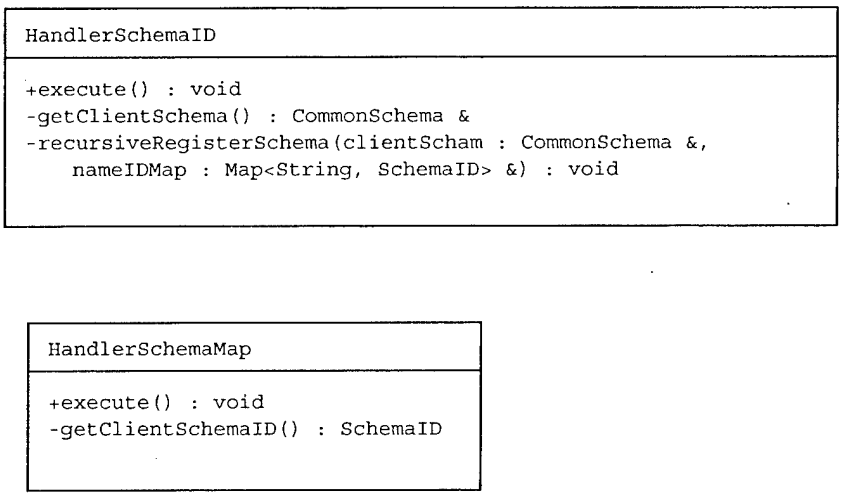


Figure 10: HandlerSchemaID and HandlerSchemaMap Class Diagram

Code Example 2 below shows the C++ implementation for the `HandlerSchemaID execute` and `recursiveRegisterSchema` methods.

```
void HandlerSchemaID::execute()
    {
    // The Map class is a templated data structure that stores a list of 1-1 mappings
    // from one type to another. Here we are mapping a class name to a schema id.
    Map<String, SchemaID> classNameMap;

    // The ReplySchemaID class's job is to send the resulting class name map
    // back to the client. Implementation will not be shown.
    ReplySchemaID reply;

    try
        {
        recursiveRegisterSchema(getClientSchema(), classNameMap);
        }
    catch (InvalidSchemaException &)
        {
        classNameMap.clear();
        reply.setError(eInvalidSchema );
        }
```

```
    reply.setClassNameMap(classNameMap);
    reply.send();
    }

void HandlerSchemaID::recursiveRegisterSchema(const CommonSchema &clientSchema,
                                Map<String, SchemaID> &nameIDMap)
    {
    SchemaID outID;

    // Check and see if the schema is already registered.
    bool found = SchemaFile::instance().lookupSchemaId(clientSchema, outID);

    // if not registered, register it.
    if (!found)
        outID = SchemaFile::instance().registerSchema(clientSchema);

    // Store the id in map
    nameIDMap.insert(clientSchema.getClassName(), outID);

    // Get the id of all the super classes
    ListIterator<CommonSchema> iter(clientSchema.getSuperClasses());
    for (iter.toFirst(); iter.current(); ++iter)
        _recursiveRegisterSchema(*(iter.current()), nameIDMap);
    }
```

Code Example 2; HandlerSchemaID Implementation

The implementation of the schema id handler class is fairly straightforward. The `execute` method creates an instance of a `Map` between class names and schema ids as shown in previous page. It then calls the protected `recursiveRegisterSchema` method to fill in the contents of the map. The map is then sent back to the client. The `recursiveRegisterSchema` method recursively works its way up the schema inheritance tree and stores the schema id for each level in the map. This method first checks to see if the schema for the current level of inheritance is registered already. If it is already registered, then it will use the existing schema id, if not, then it will obtain a new schema id by registering the schema with the `SchemaFile` singleton. Finally it will iterate through the list of super classes and call itself, passing in each super class.

Code Example 3 below shows the implementation of the `HandlerSchemaMap execute` method. This method uses the `SchemaFile::buildSchemaMap` method to get the schema map for the requested client schema id. The resulting `SchemaMap` is then sent back to the client.

```
void HandlerSchemaMap::execute()
    {
    // build a schema map for the requested schema id
    SchemaMap *map = SchemaFile::instance().buildSchemaMap(getClientSchemaID());

    // send the resulting SchemaMap back to the client
    // (or error if id not found).
    ReplySchemaMap reply;
    // the map should be valid if id was found
    if (map)
        reply.setSchemaMap(*map);
    else
        // ID not found, send back error
        reply.setError( ReplySchemaMap::eInvalidSchemaID );
    reply.send();

    // clean up the map
    delete map;
    }
```

Code Example 3: HandlerSchemaMap execute Method

Code Example 4 below shows the implementation of the `SchemaFile` class. The details of the implementation of the `loadSchema` and `saveSchema` methods are not shown. The implementation of `SchemaFile` uses a `Map` class and a `ValueList` class who's API is not shown. As discussed earlier, the Map class is a templated data structure that stores a list of 1-1 mappings between two types. The `ValueList` class is templated data structure that stores a list of type values. This class is different from a regular linked list in that it is assuming it is storing values of built in types, like `int` or `double`, as opposed to pointers to structures or objects. A `ValueList` does not have to be used, but is merely an optimization that can be made because the `SchemaID` type is just an alias for an integer. The UML for the `SchemaFile` class, shown in Figure 9 on page 20, indicates that `SchemaFile` is a singleton. This means that there will be only one instance of `SchemaFile` in the server. To implement this, a static pointer to a `SchemaFile` is defined inside the `SchemaFile` class, called `sInstance`. To access the instance, a static method called instance is defined that allocates the `SchemaFile` if needed and returns a reference to it.

```
const SchemaID SchemaFile::FIRST_SCHEMAID = 101;
SchemaFile *SchemaFile::sInstance = NULL;

SchemaFile::SchemaFile()
      : fNextSchemaID(FIRST_SCHEMAID), fClassNameMap()
    {
    }

void SchemaFile::validate(const CommonSchema &schema) const
    {
    if (schema.getClassName().isEmpty())
       throw InvalidSchemaException("SchemaFile: class name is empty");
    }

// This is a static method that implements the SchemaFile singleton.
SchemaFile &SchemaFile::instance()
    {
    if (sInstance == NULL)
       sInstance =  new SchemaFile();
    return *sInstance;
    }

SchemaID SchemaFile::registerSchema(const CommonSchema &singleSchema)
    {
    // validate the schema can be registered
    validate(singleSchema);

    // 1. Save the schema
    //     Implementation if saveSchema not shown.
    saveSchema(singleSchema);

    // 2. register in class name to schema id map
    //    a) store the new id in the class name map
    //        Get a copy of the list of schema id's for the class name
    //        from the fClassNameMap. If the specified class name doesn't
    //        already exist in the map, the [] operator will create a new
    //        entry.
    ValueList<SchemaID> idList = fClassNameMap[singleSchema.getClassName()];

    //        Store the new id in the vector.
    idList.append( registeredId );

    //        Store the vector back in the fClassNameMap
    fClassNameMap[ singleSchema.getClassName() ] = idList;

    //    b) save the class name map
```

```
    _saveClassNameMap();

    return registeredId;
    }

bool SchemaFile::lookupSchemaId(const CommonSchema &findSchema,
                        SchemaID &outId)
    {
    // validate the schema, this method will throw an exception if there
    // is a problem.
    validate(findSchema);

    // Get the list of existing schema ids for a particular class name
    // from the fClassNameMap
    ValueList<SchemaID> idList = fClassNameMap[ findSchema.getClassName() ];

    // iterate through list of schema ids
    CommonSchema *aSchema=NULL;
    for (ValueList<SchemaID>::Iterator idIter = idList.begin();
         idIter != idList.end();++idIter)
        {
        // get the schema for the current schema id.
        aSchema = loadSchema(*idIter);

        // if the schemas match, then return the schema id.
        if (aSchema->isThisLevelEqual(findSchema))
            {
            outId = *idIter;
            delete aSchema;
            return true;
            }
        delete aSchema;
        }

    // no match was found, return false.
    return false;
    }

CommonSchema *SchemaFile::loadSchema(SchemaID id)
    {
    // create a new schema to return to the caller.
    CommonSchema *schema = new CommonSchema();

    // load the contents of the schema based on the schema id.

    // implementation not shown.

    return schema;
    }

SchemaMap *SchemaFile::buildSchemaMap(SchemaID id)
    {
    CommonSchema *clientSchema = NULL;

    // try to load the schema we want to build a map for.
    // return NULL if the loading fails.
    try
        {
        clientSchema = loadSchema(id);
        }
    catch (Vortex::Exception &ex)
        {
        return NULL;
        }
```

```
    // get a list of all schemas with the same class name as the client schema
    List<CommonSchema> *serverSchemaList =
        lookupSchemas(clientSchema->getClassName());

    // create a new SchemaMap that we can add the FieldMaps to.
    SchemaMap *schemaMap  = new SchemaMap();

    // iterate through the list of schemas with the same class name.
    for (ListIterator<CommonSchema> iter(*serverSchemaList);
         iter.current();
         ++iter)
        {
        // create a field map between the client schema and the current
        // schema.
        FieldMap *fieldMap = buildFieldMap(*clientSchema, *iter.current());

        // add the fieldmap to the schema map. SchemaMap will copy the FieldMap
        schemaMap->addFieldMap(iter.current()->_getSchemaID(), *fieldMap);

        // delete the fieldMap pointer since schemaMap copied it.
        delete fieldMap;
        }

    // clean  up the client schema and the list of server schemas
    delete serverSchemaList;
    delete clientSchema;

    // return the resulting SchemaMap.
    return schemaMap;
    }

FieldMap *SchemaFile::buildFieldMap(CommonSchema &fromSchema,
                                    CommonSchema &toSchema)
    {
    // get a copy of the list of fields in the client schema.
    ValueList<SchemaField> schemaFields = fromSchema.getSchemaFields();

    // create a new field map with the number of entries matching the number
    // of fields in the client schema.
    FieldMap *fieldMap = new FieldMap(schemaFields.count());

    // iterate through all the fields in the fromSchema
    for (ValueList<SchemaField>::Iterator iter = schemaFields.begin();
         iter != schemaFields.end();
         ++iter)
        {
        // find a matching field in the toSchema
        const SchemaField *matchingField =
            toSchema.findSchemaField((*iter).getName());

        // if match was found then set the field number in the fieldMap
        if (matchingField != NULL)
            fieldMap->setField((*iter).getPosition(),
                               matchingField->getPosition(),
                               matchingField->getType(),
                               matchingField->getLength());
        else // set field number to -1 to indicate no match.
            fieldMap->setField((*iter).getPosition(),
                               -1,
                               SchemaField::eUndefinedType,
                               -1);
        }
    // return the resulting FieldMap
```

```
    return fieldMap;
    }


SchemaID SchemaFile::newKey()
    {
    // save the current key value.
    SchemaID key = fNextSchemaID;

    // increment the next key value by 1.
    fNextSchemaID++;

    // return the new key value.
    return key;
    }

List<CommonSchema> *SchemaFile::lookupSchemas(const QString &className)
    {
    // create a new list to store the resulting schemas in.
    List<CommonSchema> *schemaList = new List<CommonSchema>;
    schemaList->setAutoDelete(true);

    // find the value list associated with the class name
    Map<QString, ValueList<SchemaID> >::Iterator nameMapIter =
        fClassNameMap.find(className);

    // if not found, return empty list
    if (nameMapIter == fClassNameMap.end())
        return schemaList;

    // iterate through the value list of ids, loading each schema and
    //   storing in the out list
    for (QValueList<SchemaID>::Iterator idIter = nameMapIter.data().begin();
         idIter != nameMapIter.data().end();++idIter)
        {
        try
            {
            CommonSchema *aSchema = loadSchema( *idIter);
            schemaList->append( aSchema );
            }
        catch (Vortex::Exception &ex)
            {
            }
        }
    // return the resulting list of schemas
    return schemaList;
    }
```

Code Example 4: SchemaFile Implementation

Advantages and New Features

RASE provides several advantages over typical forms of schema evolution. By always mapping all variations of the same schema to a client's schema, older and newer client programs are allowed to co-exist in a well-defined manner. Most systems only allow one true schema, which is maintained in a central server. Evolving this schema usually involves issuing special maintenance commands to the server or writing a special client program for this purpose, which is subject to detrimental human error. Additionally, developers have to ensure that all clients are then recompiled with the latest schema. RASE eliminates these problems by allowing each client program to evolve independently, without recompiling other clients, or issuing special commands to a server.

The RASE algorithm combines flexibility with performance, by storing the object contents in a form that allows the mapping algorithm to randomly access each field. The same algorithms are used when loading objects that have matching schemas and objects whose schemas don't match. During normal use, schemas of the clients and server will match. In this case the algorithm will be able to perform a high performance direct binary load of each field of a class, without any type conversions. Additionally, the schema map for each class is usually only computed once, the first time an object of that class type is accessed. The RASE algorithm also has the advantage of keeping the server responsibilities relatively simple. The server must simply keep a list of all variations of a class, and be able to provide a map of the fields from one schema to the other. The client performs the actual work of morphing the contents of one object into another.

Most object database systems use some type of code generator that converts the developers object definitions into classes in the language they are programming in. Many of these systems involve a schema-registering phase, where a database maintenance tool is run that informs the server of the schemas the client will be using. RASE eliminates the need for registering schemas at compile time. Schemas are registered automatically, and only as needed, when the client program runs. Additionally, the process of registering a schema is very quick, since the server does not need to perform any schema evolution or mapping at that point.

In the RASE algorithm, each level of inheritance for a class is considered an independent schema. A change in the schema at one level of inheritance, does not affect the schema at a different level. This has the benefit of preventing a trickle down effect when evolving a schema. In systems that have large or deep inheritance trees, this would prevent the modification of a base class schema from causing the schema in all other objects in the system to change.

Another indirect but potentially important advantage of using the RASE algorithm and implementation is the public access to an object's schema. Providing object schema opens up the door for future uses and applications. For example, the Java language provides a form of schema access for their objects called *reflection*. Reflection is the engine behind the JavaBeans technology that allows components to be plugged together easily. JavaBeans provides a more generic ability to rapidly develop an application than exists with normal C++. By using the proposed RASE implementation, the same type of reflection provided by the Java language is then available in a C++ implementation and therefore C++Beans could be developed.

Alternatives

The code and UML examples in this paper focused mostly on implementing a system in C++. This was done for convenience, since the object database that this schema evolution algorithm was developed for was written in C++. These same algorithms could be implemented in just about any programming language, including Java, SmallTalk, and C. The RASE solution was presented in the form of an object oriented solution. However, the basic concepts could be applied in either an object oriented or procedural language.

The examples in this paper were described in terms of an object oriented database system. The RASE algorithms would work just as well for a distributed component or flat file based system. In a distributed system, an Object Request Broker (ORB) would be responsible for allowing two distributed components to talk to each other. The ORB would perform the

responsibilities of the server, keeping track of schemas as components register themselves and building schema maps for components as needed. In a flat file system, the responsibilities of the client and the server would be compiled into one program. This would allow a program to save and load objects and for those objects to evolve over time. It would not allow for a multi-user system however.

With a minimal amount of added complexity, the schema mapping algorithm can be expanded to allow the programmer to rename classes and move fields between levels of inheritance. In order to provide for renaming of fields, the programmer must provide a list of previous names of the field when defining the object in ODL. In a similar manner, the user could provide a previous name of a class, or to move a field, the previous class name and field name. The schema registering algorithm would have to be modified to handle the renaming of a class. The schema mapping algorithm would have to change to allow fields to map to a different class, and the object loading routines would have to be able to locate fields up and down the chain of `ObjectData` objects for the object being loaded.

Contributions by Inventors

The initial requirements, analysis, and design work was performed by Michael Pilone, Brian Solan and Gregory Stern in March of 2000. Brian Solan and Gregory Stern performed implementation and testing of the code, with some assistance from Michael Pilone during June of 2000. Brian Solan and Gregory Stern wrote this paper during March of 2001.

Related Publications

A. Mehta, D. Spooner, M. Hardwick. "Resolution of Type Mismatches in an Engineering Persistent Object System." Online. Internet. Rensselaer Design Research Center and Computer Science Department, Rensselaer Polytechnic Institute. 1993.

Young-Gook Ra, E. Rundensteiner. "A Transparent Object-Oriented Schema Change Approach Using View Evolution." Online. Internet. Department of Electrical Engineering and Computer Science; Software Systems Research Lab, University of Michigan, Ann Arbor. September, 1997.

K. Claypool, C. Natarajan, E. Rundensteiner. "Optimizing the Performance of Schema Evolution Sequences." Online. Internet. Computer Science Department, Worchester Polytechnic Institute. March 1999.

K. Claypool, J. Jin, E. Rundensteiner. "SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework." Online. Internet. Department of Computer Science, Worchester Polytechnic Institute. November 1998.

S. Lautemann, P. Eigner, C. Wohrle. "The Complex Object and Schema Transformation (COAST) Project: Design and Implementation." Online. Internet. Lecture Notes in Computer Science No. 1341. Montreux, Switzerland. December 1997.

G. Stern, M. Pilone, B. Solan. "A Generic Preference System Pattern and C++ Implementation." NRL/MR/5707-00-8473. ENEWS Program, Tactical Electronic Warfare Division. September 2000.

Object Design Inc. "Object Store Users Guide: DML." Twenty Five Mall Road, Burlington, MA 01803. 1993.

Objectivity Inc. "Objectivity Technical Overview. " 301B East Evelyn Ave. Mountain View, CA 94041. April 1996.